PARALLEL FUNCTION PROCESSOR
PROGRAMMER'S MANUAL

SPECIAL TECHNICAL REPORT

REPORT NO. STR-0142-90-006

January 23, 1990

GUIDANCE, NAVIGATION AND CONTROL
DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332 - 0540

Contract Data Requirements List Item A004

Period Covered: Not Applicable

Type Report: As Required

DISCLAIMER

DISCLAIMER STATEMENT - The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

DISTRIBUTION CONTROL

(1) DISTRIBUTION STATEMENT - Approved for public release; distribution is unlimited.

(2) This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227 - 7013, October 1988.

PARALLEL FUNCTION PROCESSOR
PROGRAMMER'S MANUAL

January 23, 1990

Author

Richard M. Pitts

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology

Atlanta, Georgia 30332-0540

Eugene L. Sanders

USASDC

Contract Monitor

Cecil O. Alford

Georgia Tech

Project Director

Table of Contents

List of Tables

1. SCOPE

1.1 Identification

This Software Programmer's manual applies to the Georgia Tech Parallel Function Processor (PFP), Georgia Tech part number CERL002-0757-000.0. The Parallel Function Processor (PFP) hardware and software are partitioned into the following two categories; host and target.

The host computer hardware consists of an Intel 310, terminal, and printer. The target computer hardware is the main PFP system unit consisting of thirty-two processing elements, a 16x16 crossbar, the crossbar sequencer, and associated interconnections. The processing elements are generally single board computers (SBC), and are referred to as target processors. The PFP can be upgraded to two clusters of thirty-two processing elements, two crossbars, and two crossbar sequencers.

The system software is divided into two sections; host software and target software. Software which executes on the host during a simulation is referred to as the host program. Target software, which is executed on the PFP, is divided into three sections. Programs executed on target processors are called processor code. The microcode loaded into the sequencer memory is called sequencer code. The microcode loaded into the crossbar memory is called crossbar code. All target software is written and compiled at the host, then downloaded to the PFP for execution.

1.2 System Overview

The purpose of the Parallel Function Processor is to solve systems of differential equations in real-time via the parallel architecture of the machine. The crossbar communication structure between processors facilitates this by allowing a sequence of flexible and dynamic communication events to occur. Each processor can be assigned one or more differential equations (states) to solve. Using the crossbar, state information can be communicated between the processors simultaneously so that the solution is calculated fast and accurately. Not only is the PFP well suited for solving systems of differential equations it is also appropriate for many other programs that can be partitioned into modules, where all communication paths and data transfer lengths are known in advance.

1.3 Document Overview

This document contains the information for a programmer to understand and program the Parallel Function Processor. Information on languages, syntax, and memory limits will be presented. Additional information on how to use existing system software is discussed.

2. REFERENCED DOCUMENTS

The following documents contain information which is useful in unerstanding the PFP hardware and software. These documents should be consulted for additional details on specific issues.

Intel iRMX Reference Manuals Vol. 1-7
Intel 80286 SBC Hardware Reference Manual
Intel 80386 SBC Hardware Reference Manual
Intel 215 Disk Controller Hardware Reference Manual
Intel Pascal Software Reference Manual
Intel FORTRAN Software Reference Manual
Intel C Software Reference Manual
Georgia Tech PFP Technical Data Package
PFP Hardware Operation Manual

3. SOFTWARE PROGRAMMING ENVIRONMENT

3.1 Equipment Configuration

3.1.1 Intel 310 Host Computer

The host computer consists of an Intel 310, Multibus I based computer, a 40 Mbyte fixed disk drive, a 360K floppy disk drive, and a terminal. The Intel 310 computer is a 80286 based computer running the iRMX operating system. The host serves as the platform for software development and as the interface to the PFP. A Multibus I repeater system is used to interconnect the host to the PFP and all communication between the host and each processing element is accomplished via the Multibus repeater system.

3.1.2 PFP Target Computer

The PFP consists of 32 processing elements, one crossbar, and one sequencer and can be upgraded to 64 processing elements, two crossbars, and two sequencers. The host can access all of these through the Multibus I repeater system. The processing elements are usually single board computers but can be other items such as an array interconnect board or an analog I/O board. The crossbar is the dynamic switch that allows flexible data communications between the processing elements. The sequencer is the device that controls the crossbar switching based on an apriori sequence of instructions describing the set of communication patterns to be performed between the processing elements during a simulation. Each processing element has a Multibus I interface and a sequencer/crossbar interface. Interfacing to the PFP can be done either through the Multibus I port and/or a crossbar/sequencer port.

3.2 Operational Information

3.2.1 Intel 310 Host Computer

For more in depth coverage of the features of the host refer to the Intel iSBC286/12 Hardware Reference Manual and the Parallel Function Processor Operation Manual.

3.2.2 PFP Target Computer

The PFP configuration uses several types of processing elements. The Intel 286/12 and 386/12 are commercially available single board computers. Other processing elements have been developed by Georgia Tech as special purpose, high performance processors.

3.2.2.1 Intel Single Board Computer

Refer to the Intel iSBC286/12 Hardware Reference Manual for detailed information. The 80286 single board computer contains 1 Megabyte of memory of which 64 Kbytes is accessible via the Multibus repeater

system. The 286/12 uses an 80287 co-processor. Clock frequency is 8MHz. Refer to the Intel iSBC286/12 Hardware Reference Manual for detailed information.

The 80386 single board computer contains 1 Megabyte of memory of which 64 Kbytes is accessible via the Multibus repeater system. The 386/12 uses an 80387 co-processor. Clock frequency is 12MHz. Refer to the Intel iSBC386/12 Hardware Reference Manual for detailed information.

The GT-FPP/3 single board computer contains 4K of 96 bit wide instruction memory and 2K of 32 bit wide data memory. All instruction memory is accessible via the Multibus repeater system. The FPP uses an AMD 29C325 processor. Clock frequency is 10MHz. The GT-FPP/3 has a throughput of 8 MFlops. Refer to the Georgia Tech GT-FPP/3 Hardware and Software Reference Manuals for detailed information.

3.2.2.2 Sequencer and Crossbar

The sequencer and crossbar work in conjunction with the processing elements to yield inter-processor communication. A sequence of communication patterns described in a crossbar input file is used to generate microcode for the crossbar and sequencer. The crossbar microcode defines which paths on the crossbar are connected and the sequencer microcode selects which processing elements will be involved during a particular communication cycle, waits for the appropriate status flags, generates the data transfer signals, and then advances to the next communication cycle. Refer to the Georgia Tech GT-SEQ/2 Sequencer Design section and to the GT-XB/2 Crossbar Design section of the Georgia Tech PFP Technical Data Package.

3.3 Compiling, Linking and Running

The PFP has up to two clusters of thirty-two processing elements (usually processors). The thirty-two processing elements in a single cluster communicate with each other over one 16x16 crossbar. The Multibus I bus repeater system is used by the host to communicate with the PFP. The host computer is used to generate the object code for the processing elements. At run-time the host computer performs downloading and starts the target processors.

The basic concepts are:

i) During a simulation each active processing element performs a function (a computer program) and, when necessary, sends data to or receives data from other processing elements over the crossbar.

ii) All processors are downloaded with a program using an input file containing a list of the processors and the programs to be loaded into the processors. Some processing elements like the GT-ADDA/2 analog I/O board do not 'run a program' but have the ability to perform any required communcation and processing.

iii) During a simulation the majority of the communication between processing elements takes place across the crossbar, although inter-processor communication can occur via the host computer.

iv) The desired direction for communication between processing elements must be allowed by all the involved elements. Processors request this through a program statement which controls communication direction (e.g., send, receive). All processors which are involved with a given communication must allow the requested transfer.

v) The crossbar/sequencer combination must know and allow the desired communication; (this control is handled through the crossbar definition file, the '.xbc' file).

After the required programs are written, the following items must be done before 'run time':

1) compile host program
   - **submit :PFP:csd/hPascal(-)**
   - **submit :PFP:csd/hFortran(-)**
   - **submit :PFP:csd/hC(-)**

2) compile processor code
   - **submit :PFP:csd/fortran(-)**
   - **submit :PFP:csd/pascal(-)**
   - **submit :PFP:csd/c(-)**

3) compile crossbar code (if using the crossbar)

   - **xbc**

Note: Compiling the host program and the processor code requires several very long commands. These commands have been placed in command files for the different languages. The commands used to invoke these command files are listed above with the appropriate sections.

At run time the following must be performed:

1) reset the system

   - **reset**

2) reset the crossbar (if used)

   - **resetxbar**

3) load the processors

   - **simload filename.ext** (filename.ext = name of the processor list file)

4) load the sequencer and crossbar (if used)

   - **sxbload 1**

5) start the sequencer (if used)

- **masseq 1**

6) start the host program (which starts the processors)

- **filename** (filename = name of the host program)

Note: The above items 1-6 will generally be contained in a command file. This command file will be called Run.csd for a program using the crossbar or Runnc.csd for a program that does not use the crossbar. These command files can be executed by typing '**submit :PFP:csd/run**' or '**submit :PFP:csd:/runnc**'.

Note: The order of the above can be changed. In general, the resetting of a device precedes the loading of the same device and loading precedes starting. The sequencer is usually started before the host program starts, but this is not mandatory.

3.3.1 Intel 310 Host Computer

3.3.1.1 Compiling

For a comprehensive approach to compiling a program refer to the appropriate Intel language manual listed in section 2 of this document. Complete examples are in Appendix B. Some brief examples are:

FORTRAN:

- fort86 file_name.ext (.ext = .for)

Pascal:

- pasc86 file_name.ext (" .pas)

PLM:

- plm86 file_name.ext (" .plm)

C:

- ic86 file_name.ext (" .c)

Note: No extensions are assumed for the compilers (i.e., if a FORTRAN file was named 'test.for', 'fort86 test.for' NOT 'fort86 test' is required. If a FORTRAN file is named test.src, 'fort86 test.src' must be used. However the output file for the compiler always replaces the ext(ension) of the input file name with '.obj' (e.g., 'fort86 test.for' outputs 'test.obj'). The compilers also generate a compiled listing with the extension '.lst'. See the appropriate compiler manual for controls that affect the list file and code generation.

3.3.1.2 Linking

For a comprehensive look at linking refer to the Link Utility in the iAPX 86, 88 Family Users Guide, in Vol 7, Languages and Utilities, Part 2 of the Intel iRMX manual set and the examples in Appendix B. The following is a brief summary of the items that need to be addressed.

In the iRMX environment all references to routines that are not defined within the code written by the user have to be resolved at link time. Although this is not uncommon, in addition to any user defined external references there are multiple language dependent and independent libraries that have to be identified and linked into the program such as co-processor libraries. Another area to be addressed is that of making the linked code into a host executable image. This is accomplished by the BIND option of the link command. Example:

```
- link86 test.obj, fortran.lib to test bind
```

This links test.obj and library fortran.lib to an output file named 'test' and the bind option makes it a host executable image file.

3.3.1.3 Running

To run a file that has been compiled and linked simply type the executable image file name

```
- test
```

and the program will run. Refer to Appendices A and B for a brief overview of I/O control at the iRMX level.

3.3.2 PFP Target Computer

3.3.2.1 Target Processor

3.3.2.1.1 Compiling

See as section 3.3.1.1

3.3.2.1.2 Linking and Locating

For a comprehensive look at linking refer to the Link Utility in the iAPX 86, 88 Family Users Guide, in Vol 7, Languages and Utilities, Part 2, of the Intel iRMX manual set and the examples in Appendix B. The following will give a brief summary of the items that need to be addressed.

In the iRMX environment all references to routines that are not defined within the code written by the user have to be resolved at link time. Although this is not uncommon, in addition to any user defined external

references there are multiple language dependent and independent libraries that have to be identified and linked into the program such as co-processor libraries. Example:

- link86 test.obj, fortran.lib to test.lnk

This links test.obj and library fortran.lib to an output file named 'test.lnk'.

The linker output file 'test.lnk' now needs to be located with the loc86 utility (See Loc86 in the iAPX 86, 88 Family Users Guide, in Vol 7, Languages and Utilities, Part 2, of the Intel iRMX manual set). The locater utility allows the adjustment of the default run time allocations. Specifically, stack size and reserved memory locations that are not to be loaded must be controlled. Example:

```
- loc86 test.lnk to test.abs &
  segsize(stack(+100h)) &
  reserve( 0H to 007FFH, 0D0000H to 0FFFFFH )
```

where the object file 'test.lnk' generates the located file 'test.abs' with 100H bytes of additional stack space compared to the default model, and does not use memory locations 0H to 07FFH and 0D0000H to 0FFFFFH.

More comprehensive examples are in Appendix B. Now that the code has been located an offset adjustment has to be performed on the located code to allow for proper loading onto the target processor. This is accomplished by

- fixomf86 test.abs e0

This adds an offset of e0000H to the absolute loader records of the binary file 'test.abs' and the output file is named 'test.fix'. This offset adjustment is required because the host computer memory addresses are mapped into 64K segments on processing elements and are accessed through memory location e0000H.

3.3.2.1.3 Loading

To load a compiled, linked, located, and fixed file the utility simload is used and is invoked by:

- simload filename.ext

where filename.ext is the name of the simload input file with lines containing a processor designator in the set {x0,x1,...,x15,y0,y1,...,y15} and the name of the file to be loaded to the processor. The simloard program loads the file to the designated processor. Example input file:

```
y0 test1.fix
x0 test2.fix
x5 output.fix
```

The simload program would load the file 'test1.fix' to processor y0 and so forth.

Note: An error will occur at run time if the last line of the simload input file does not have a return at the end of the line. Also, an error will occur during the execution of simload if the last line of the simload input file is blank.

3.3.2.1.4 Running

A host program must be executed in order to start the loaded target processors. The host program must read in the processor list file and match processors to actual hardware addresses. The routine 'initialize_proc_info' is invoked to accomplish these two functions. This routine inputs information into an array named 'processor'. The routine 'start_all_processors' is then called to start all of the processors involved in the run and then the routine 'set_up_host_io' is called to initialize the status words for the active processors. See section 4.1.5 for examples and Appendix C for type definitions.

Note: The language for the host needs to be able to support a structured type of memory access. The preferred language for the host is currently Pascal. (See the Pascal 'proc_info' type definition in the 'sysinfo.def' procedure definition of Appendix C for the structure organization.)

The host program must also handle any input or output needed from target processors during the simulation.

3.3.2.2 Crossbar and Sequencer

The crossbar and sequencer microcode are both generated by the crossbar compiler. The input to the crossbar compiler is a crossbar definition file, which describes the communication patterns for each crossbar cycle.

3.3.2.2.1 Compiling

Assume a file named 'example.xbc' has been created and contains communication definition statements. The file would be compiled by entering 'xbc' at the iRMX prompt and then responding with 'example.xbc' for the input file name and then 'n' for the two follow up questions.

- xbc

Enter the communication file name - **example.xbc**

Do you want setup maps - **n** (generates SETUP.DAT)

Do you want address maps - **n** (generates ADDR.DAT)

If an 'n' is given in response to the latter two questions, the crossbar compiler (xbc) generates two files named 'seq.abs' and 'xbar.abs'. The former is the microcode for the sequencer and the latter is the microcode for the crossbar. If a 'y' is given in response to the latter two questions the files setup.dat and addr.dat are generated and can be used for debugging.

The normal output to the terminal for the crossbar compiler is to list a sequence denoting which cycle is being processed. Errors are written to an 'error.dat' file.

3.3.2.2.2 Linking and Locating

No linking or locating is required after compiling crossbar and sequencer code. The file is ready to load.

3.3.2.2.3 Loading

To load the files generated by the crossbar compiler, a utility named 'sxbload' is used. This utility assumes that the files seq.abs and xbar.abs exist and will load them into the sequencer and crossbar. In addition to these two files, the sequencer/crossbar loader needs to know the system definition number of the sequencer. (The system is capable of handling two sequencers which are usually numbered 0 and 1, although other numbers can be used.)

Example:

- sxbload 1

would attempt to load seq.abs and xbar.abs to unit 1's sequencer and crossbar, respectively.

Also, alternate names for the input files can be used as in the case where the seq.abs and xbar.abs files have been renamed.

Example:

Assume: seq.abs/xbar.abs are now RK4.seq/RK4.xbar

- sxbload 0 RK4.seq RK4.xbar

would attempt to load RK4.seq and RK4.xbar to unit 0's sequencer and crossbar, respectively.

3.3.2.2.4 Running

To start the sequencer, the utility 'masseq' is invoked. A unit number is used to identify which unit is to be started.

Example:

- masseq 0 (starts unit 0)

- masseq 1 (starts unit 1)

Multiple crossbars and sequencers are loaded and started individually.

4. PROGRAMMING INFORMATION

4.1 Host Computer

The Intel 310 host is a commercially available computer with special functions and procedures written by Georgia Tech to enable it to communicate with the PFP. Refer to the iRMX manual set, the iSBC286/12 hardware reference manual, and the appropriate language manual for comprehensive information. Appendices C, D, and E list the special functions and procedures available to communicate with the PFP. The following paragraphs give a general discussion of the more important topics.

4.1.1 Programming Features

Refer to the appropriate Intel laanguage reference manual listed in Section 2 of this document for the data types supported by a particular language. See Appendix C for the Pascal I/O routines used to communicate between the host and the target processors.

4.1.2 Programming Instructions

See the appropriate Intel language manual(s) listed in Section 2 of this document.

4.1.3 Input and Output Control Programming

Two serial ports and one parallel I/O port are available on the single board computer module in the Intel 310. The terminal is connected to one serial port and a printer may be connected to the parallel port. Additionally, an iSBX expansion port is available on the single board computer. The host computer is interfaced to the PFP through the Multibus. This interfacing is accomplished via a repeater scheme where the host computer utilizes a 'host repeater' board and the PFP's Multibus card cages use 'slave repeater' boards. Low-level program routines access this repeater system in order to pass data between the host and target computers. All standard I/O routines such as reads, writes, and file manipulation utilities are available and can be utilized.

4.1.4 Additional or Special Techniques

The low-level utilities for communicating between the host and the PFP are addressed in this section.

The repeater system that is used to communicate between the host and the PFP is based on a memory window scheme. In order to access more than the 16 Megabyte address space of the Multibus I specification, a paging scheme was developed and implemented on the PFP. The paging scheme allows window sizes of 64KB or 1MB depending on the need. The use of a window is accessed by 'turning on' a repeater at a specific 64KB or 1MB boundary with the turn_on_repeater rountine described in Appendix C. After the repeater is turned on, the memory of the target processor at that window appears as an I/O port to the host and is accessed as such via the send and receive routines listed in Appendices C, D, and E. The

low level routines are used on both the target and the host in order to achieve a data transfer. After the required transfer is completed the repeater is 'turned off' with the turn_off_repeater rountine described in Appendix C..

This procedure of turning the repeater on, communicating, and then turning the repeater off is much like the process of making a telephone call, trading the required information, and then hanging up. Each window represents one target processor. Before starting the next communication sequence with another target processor it is required to complete the above sequence by turning the repeater off.

Listings of special I/O functions and procedures are given in Appendix C.

4.1.5 Programming Examples

Program Listing 1. Host Program Model

```
module main;
{This is an example host program written in Pascal. This program is not complete in that
all of the procedures referenced are not defined. It shows the structure of a typical host
program }

{This program scans the target processor checking if any of the processors wish to
communicate with the host. If a processor is trying to communicate with the host, the host
expects the processor to send a variable name and then the value of the variable. Then
the host prints the variable name and value to the PRINT.LOG file and starts scanning
the processors again. If the host scans the processors more than 10,000 time in a row the
program will end

$include(':PFP:INCLUDE/sysinfo.def')
$include(':PFP:INCLUDE/multiio.def')
$include(':PFP:INCLUDE/startproc.def')
$include(':PFP:INCLUDE/com.def')

program main(input,output,print,proc_list);

const
        string_length = 32;

type
        string_type = array [1..string_length] of char;

var
   print: text;
   proc_list: text;


procedure input_and_print( var output: text; maximum_count: integer );

var i: integer;
   count: integer;
   string: string_type;
```

```
  message_size: word;
  message: generic_type;
  message_type: word;

begin
        count := 1;
        while ( count <= maximum_count ) do
        begin
                for i := 1 to Number_of_proc do
                begin
                        {Connect host to processor i for communications. i = position
                        in processor list file}

                        turn_on_repeater(processor[i].page_port, processor[i].page );
                        {Check if processor is ready to communicate.)

                        if ( io_ready )
                        then
                                begin
                                        count := 1;

                                        {Read variable name}
                                        input_message(message_type,
                                           message, message_size);
                                        make_string(string, message,  message_size);

                                        {Read variable value}
                                        input_message(message_type, message,
                                                message_size);

                                        {Print variable and its value to Print.log}
                                        print_message(output, string, message_type,
                                                message, message_size );
                                end;
                                {Disconnect host from processori}
                                turn_off_repeater( processor[i].page_port );
                end;
                count := count + 1;
        end;
end; { input_and_print }

begin
   {Stuff processor array structure with processors from the processor list file}
   initialize_proc_info(proc_list);

   {Clear all IO channels between host and target procssors}
   set_up_host_io;

   {Start all of the processors in the processor list file}
   start_all_proc;

   rewrite(print);
   input_and_print(print, 1000);
end.
```

4.1.6 Error Detection and Diagnostic Features

Compilation and linking errors are written to the terminal and to the appropriate listing file (.lst for compile or .mp1 for link). Run time errors are displayed on the terminal. Refer to the Intel iRMX manuals and Intel language manuals for explanations of the errors.

4.2 Target Computer

4.2.1 Target Processors

The Intel iSBC286/12 is a commercially available single board computer. Refer to the iRMX manual set, the iSBC286/12 hardware reference manual, and the appropriate language manual for comprehensive information. The following paragraphs give a general discussion of the more prominent information.

4.2.1.1 Programming Features

Refer to the appropriate Intel language reference manual listed in Section 2 of this document for the data types supported by a particular language. See Appendices C, D, and E for the I/O routines supported for Pascal, FORTRAN, and C, respectively.

4.2.1.2 Programming Instructions

See the appropriate Intel language manual(s) listed in Section 2 of this document.

4.2.1.3 Input and Output Control Programming

Two serial ports and one parallel port are available on each processor for connection to external devices. One of the serial ports can be utilized by a set of instructions given in Appendices C, D, and E for Pascal, FORTRAN, and C, respectively. Additionally, an iSBX expansion port is available on the board. The target processors have an interface with both the host computer via the Multibus and with the crossbar network through the iSBX port. No standard I/O routines such as reads, writes, and file manipulation are available on the target processors. Only the low-level routines to communicate to the host via the Multibus and the crossbar via the iSBX port are supported.

4.2.1.4 Additional or Special Techniques

The low-level utilities for communicating between the host and the PFP are addressed in this section. See section 4.1.4 for more background on the repeater system for Multibus communications between the target processors and the host.

The target processors can communicate to either the host or to another target processor. Communication with the host is accomplished through the multibus repeater system. Communication to the host across the Multibus can only be performed when the host has accessed a specific processing element (i.e., when the host has turned on the memory page to the element.) In order to transfer data the target processor issues a send for every receive issued by the host, and a receive for every send issued by the host.

The target processors communicate with each other by issuing sends and receives via the iSBX port to the crossbar. The other processors involved in a crossbar communication cycle are required to accept or generate data appropriately. Additionally, as covered in section 4.2.2, the crossbar and sequencer will need to be programmed to accommodate these communication patterns.

4.2.1.5 Programming Examples

The following is a target processor's program to communicate with the host:

Program Listing 2. Example Target Processor Model - A

```
module main;

$include(':PFP:INCLUDE/com.def')

program main;

var

  r32: array [1..2] of REAL_32BIT_type{real};
  s16: array [1..2] of SIGNED_16BIT_type{integer};

begin
        r32[1] := 1.0;
        r32[2] := 2.0;
        s16[1] := -3;
        s16[2] := -4;

        output_message( CHARACTER_08BIT, 'r32', 3 );
        output_message( REAL_32BIT, r32, 2 );

        output_message( CHARACTER_08BIT, 's16', 3 );
        output_message( SIGNED_16BIT, s16, 2 );

end.
```

The following three examples are target programs in Pascal, FORTRAN, and C that perform some simple host to target communication and some inter-processor communication.

Program Listing 3. Pascal Target Processor Model - B

```
module main;

$include(':PFP:INCLUDE/com.def')

program main;

var

        r32: array [1..2] of REAL_32BIT_type;
        s16: array [1..2] of SIGNED_16BIT_type;

begin
        receive_REAL_32BIT( r32[1] );
        receive_REAL_32BIT( r32[2] );

        receive_SIGNED_16BIT( s16[1] );
        receive_SIGNED_16BIT( s16[2] );

        output_message( CHARACTER_08BIT, 'r32', 3 );
        output_message( REAL_32BIT, r32, 2 );

        output_message( CHARACTER_08BIT, 's16', 3 );
        output_message( SIGNED_16BIT, s16, 2 );
        r32[1] := 4.0;
        r32[2] := 3.0;

        s16[1] := -4;
        s16[2] := -3;

        output_message( CHARACTER_08BIT, 'r32', 3 );
        output_message( REAL_32BIT, r32, 2 );

        output_message( CHARACTER_08BIT, 's16', 3 );
        output_message( SIGNED_16BIT, s16, 2 );

        send_REAL_32BIT( r32[1] );
        send_REAL_32BIT( r32[2] );

        send_SIGNED_16BIT( s16[1] );
        send_SIGNED_16BIT( s16[2] );

end.
```

Program Listing 4. Fortran Target Processor Model

```
        program main

$include(':PFP:INCLUDE/com.inc')

        REAL*4 r32(2)
        INTEGER*2 s16(2)

        r32(1) = 1.0
        r32(2) = 2.0

        s16(1) = -3
        s16(2) = -4

call output_message( %VAL(CHARACTER_08BIT), 'r32' )
call output_message( %VAL(REAL_32BIT), r32, %VAL(2) )

call output_message( %VAL(CHARACTER_08BIT), 's16' )
call output_message( %VAL(SIGNED_16BIT), s16, %VAL(2) )

call send_REAL_32BIT( r32(1) )
call send_REAL_32BIT( r32(2) )

call send_SIGNED_16BIT( s16(1) )
call send_SIGNED_16BIT( s16(2) )

call receive_REAL_32BIT( r32(1) )
call receive_REAL_32BIT( r32(2) )

call receive_SIGNED_16BIT( s16(1) )
call receive_SIGNED_16BIT( s16(2) )

call output_message( %VAL(CHARACTER_08BIT), 'r32' )
call output_message( %VAL(REAL_32BIT), r32, %VAL(2) )

call output_message( %VAL(CHARACTER_08BIT), 's16' )
call output_message( %VAL(SIGNED_16BIT), s16, %VAL(2) )

        end
```

Program Listing 5. C Target Processor Model

```
#include <com.h>

main( )
{
        REAL_32BIT_type r32[2];
```

```
    SIGNED_16BIT_type s16[2];

    receive_REAL_32BIT( &r32[0] );
    receive_REAL_32BIT( &r32[1] );

    receive_SIGNED_16BIT( &s16[0] );
    receive_SIGNED_16BIT( &s16[1] );

    output_message( CHARACTER_08BIT, "r32", 3 );
    output_message( REAL_32BIT, r32, 2 );

    output_message( CHARACTER_08BIT, "s16", 3 );
    output_message( SIGNED_16BIT, s16, 2 );


    r32[0] = 4.0;
    r32[1] = 3.0;

    s16[0] = -4;
    s16[1] = -3;

    output_message( CHARACTER_08BIT, "r32", 3 );
    output_message( REAL_32BIT, r32, 2 );

    output_message( CHARACTER_08BIT, "s16", 3 );
    output_message( SIGNED_16BIT, s16, 2 );

    send_REAL_32BIT( &r32[0] );
    send_REAL_32BIT( &r32[1] );

    send_SIGNED_16BIT( &s16[0] );
    send_SIGNED_16BIT( &s16[1] );

} /* main */
```

4.2.1.6 Error Detection and Diagnostic Features

Compilation, linking, and locating errors are written to the terminal and to the appropriate listing file (.lst for compile, .mp1 for link, or .mp2 for locate). Refer to the Intel iRMX and language manuals for explanations. Run time errors will be determined by the correctness and the completeness of an execution sequence.

4.2.2 Crossbar and Sequencer

The crossbar and sequencer work as a unified system. The crossbar compiler (xbc) generates microcode for the crossbar and the sequencer from an input file that describes the communication cycles and patterns to be implemented during the execution of a multi-processor program. The sequencer controls or as the

name implies, sequences the crossbar through the defined data path connections described by the input file. Each crossbar and sequencer combination supports 32 processing elements and allows for multiple sets of communications to occur at the same time. A communication cycle is one processing element transferring data to one or more processing elements.

4.2.2.1 Programming Features

The communication paths for the crossbar are 16 bits wide. Thus all communication between processors is performed as sets of 16 bit transfers (e.g., a 16 bit integer is transferred in one transfer cycle, a 32 bit floating point number is transferred in two cycles.)

Single instructions allow the transfer of data from one processing element to one or more processing elements. The sequencer and compiler combination supports two constructs or flow control statements: (i) CYCLE and (ii) LOOP. The CYCLE construct allows for the grouping of one or more single instructions into one simultaneous communication. The CYCLE construct allows for parallel communications to occur. The LOOP construct is used to define a set of CYCLEs that are to be repeated indefinitely.

An example of this would be a simulation that requires some initial data transfer to initialize all the state variables and then to begin an integration routine where a set of variables needs to be communicated during each integration step. The initial data transfer requires a CYCLE construct which is executed once. The integration requires a CYCLE construct which is executed repeatedly using the LOOP construct.

Comments are opened by a '[' and closed by a ']' and cannot be nested. The CYCLE construct groups the single instructions between the current CYCLE statement and the next CYCLE statement. By definition the CYCLE constructs cannot be nested. The LOOP construct can only be used once per input file. LOOP lumps all the CYCLE statements that follow it into one big loop (i.e. all statements between the LOOP statement and the end of the file are grouped together and are repeated indefinitely.)

4.2.2.2 Programming Instructions

There is only one instruction. The transfer instruction has the following syntax:

$P_i,P_k,\cdots,P_j := P_l[.n][;]$

Processor P_l is transferring data to the set of processors $P_i,P_k,\cdots,P_j$. The number of 16 bit transfers is controlled by the .n option where n is an integer. The ';' is an optional line terminator. If the .n option is omitted the default is n=1. If a 32 bit value is to be transferred, the n is replaced by a 2 (e.g., $:= P_l.2$). The set of processors $P_i,P_k,\cdots,P_j$ does not have to be monotonically increasing or decreasing, but for clarity it is helpful if the sequence is monotonic. All subscripts for the P(rocessor) numbers are in the set [0,31]. There is a one to one mapping between the set of P numbers [0,31] and the set of processor identification numbers {x0,x1,....,x15,y0,y1,....,y15} in section 3.3.2.1.3. The mapping is item for item with respect to the ordinal number of the sets (e.g., P0-x0, P15-x15, P16-y0, and P31-y15). The processor id appearing on the

right side of the transfer equation cannot appear on the left. It also follows that a processor id cannot appear more than once during a cycle.

4.2.2.3 Input and Output Control Programming

The communication between the sequencer and the host is accomplished with the same repeater scheme used for the target processors. All work is done by the host and the sequencer is used as a bank of memory until the I/O start command is issued. At this time the sequencer begins sequencing the communication activities. The crossbar is accessed through the sequencer and is controlled as a memory bank, similar to the sequencer, via the repeater system.

4.2.2.4 Additional or Special Techniques

Since the communication patterns have to be determined apriori, all possible data transfers between processing elements must be described in the crossbar input file. This requires that the processing elements generate and accept transfers during each loop of the LOOP code. The processing elements will know or determine which datums are valid and which are not valid.

4.2.2.5 Programming Examples

The following is an example crossbar definition file.

Program Listing 6. Example Crossbar File

```
[ Crossbar for Test 3 ]

   CYCLE                                  [ initialize ]
        p4,  p5,  p9  := p15.2
        p2 := p3.2
LOOP                                            [ main loop ]
   CYCLE
        p1,  p2,  p3  := p0
   CYCLE
        p4,  p5,  p9  := p15.2
        p0,  p2,  p3  := p1
   CYCLE
        p0,  p1,  p3  := p2
   CYCLE
        p0,  p1,  p2  := p3
```

4.2.2.6 Error Detection and Diagnostic Features

The normal output to the terminal for the crossbar compiler is a numerical sequence denoting each cycle as it is being processed. Compilation errors are written to an 'error.dat' file as they are encountered during

compilation and are self-explanatory. Some common errors are: (i) using periods instead of commas, (ii) use of the same processor id on both sides of the ':=', and (iii) the same processor id used more than once during a CYCLE. Run time errors will be determined by the correctness and the completeness of an execution sequence.

5. NOTES

Programming in a parallel environment requires considerations not encountered in a serial environment. Since there are multiple processes occurring simultaneously, coordination of communication and computation for each process is required. Data dependencies between processes have to be such that one processor is not expecting data from another processor which in turn is expecting data from the former. This state is referred to as deadlock. Deadlock can also happen between more than two processes.

Data to be transferred from one process to another requires a communication path. When the programmer is scheduling the inter-processor communication (i.e., writing the crossbar code) attention has to be given to determine if the desired number of transfers can be performed during one cycle. Depending on the number of transfers already scheduled in a specific cycle, the addition of another data transfer may need to be delayed until the next available cycle, unless one of the currently scheduled data exchanges can be moved.

Communication between multiple target processors, or between target processors and the host, have to be coordinated so that each process sending information has a process to correctly receive each datum being sent (and vice versa). These communications are matched according to data type (real, integer, etc.). Inter-processor communications also requires matching sequencer and crossbar code (the inter-processor communication file) in order for target processors to communicate, in addition to the sends and receives on the processors.

Sending control information to processes, and debugging in a parallel environment, involves the receiving and sometimes the sending of messages to the individual processes during a simulation. Each of these messages must be tagged or labeled appropriately so a thorough understanding of the information is easily achieved. On the PFP, all interaction with the processes by the operator is accomplished via the host. Interaction implies sending a (data) message to the processor or receiving a (data) message from the processor. As an example, if a process needs to send out a message to the operator, the process sends the message to the host and then the host either displays it at the terminal or stores it in a file. Other forms of displaying information by the processes can be through analog I/O or through the control of status LEDs on the individual processors. This process is analogous to having to perform all I/O for a standard program in the main program, with information being generated in or transmitted to subprograms, as opposed to being able to do at will the desired reads and writes in the subprograms.

Appendices C, D, and E contain the message passing information for both processor to host communication and processor to processor communication for Pascal, FORTRAN, and C respectively.

6. APPENDIX

6.1 APPENDIX A

SUMMARY OF IRMX COMMANDS

Operating system : iRMX

Intel iRMX is not case sensitive, except in the case of the account password.

Some useful commands:

- submit file_nam

runs a command file with name fil_nam.csd

A command file is a text file with a series of iRMX commands that are to be executed together repetitively. The .csd file is very similar to the .bat file in DOS and the .com file in VAX/VMS.

- submit hostio(hostfile)

runs command file hostio with parameter 'hostfile'

- submit file_nam to :LP:

runs command file file_name and sends output to :LP:

- submit file_name to out_name echo

runs command file and sends output to file out_name and echos it to the screen

- attachfile path (or **cd path**)

changes directory to path (if it exits)

Examples this and other uses of cd:

cd	
cd /pfp/test	
cd /	(moves to root directory)
cd ^	(moves to parent directory))
cd :sd:	(moves to root directory of the system disk)
cd :w1:	(moves to root directory of the user disk)
cd $ as :a_name:	(assigns :a_name: to the current directory)
cd :a_name:	(moves to directory specified by the logical :a_name:)

cd :sd:6dof/kwest to :kwest:

cd :sd:6dof/kwest

- createdir subdir

makes directory as specified by subdir

- copy fil_name(s)

displays the contents of said file on the screen

- copy [path]file1 to [path]file2

obvious

- copy file(s) to :lp:

prints file to printer

- copy file(s) to :$:

copies file(s) to current directory

- dir (or dir :$: or dir $)

directory

- dir :PFP:

- dir /system

- dir $ for *dat

directory of files ending in 'dat' in the current directory :$:

- dir $ to :LP: for *.dat

sends directory of :$: to printer :LP:

- attachdevice wmf0 as :f:

allows the floppy in the floppy drive to be used as device :f: See detachdevice :f:

- detachdevice :f:

removes the floppy from use - must be used before removing floppy!

- attachdevice wta0 as :t: physical

allows the tape in the drive to be used as :t:

- detachdevice :t:

removes the tape from use

- delete file(s)

obvious, works for empty and unprotected directories also

- permit file_name(s) drau user=1

sets common file access privilege to 'file_name'

(e.g., **- permit RK4.for drau user=1**

- permit test/* drau user=1
- permit * drau user=1)

- path

displays current directory path (e.g., /spock/test)

- ^R

recalls previous commands (can not go forward list)

- ^C

cancels current command

- backup :sd: to :t:
- backup :w1: to :t:

backs up the files on logical names :sd: and :w1: respectively. :sd: and :w1: are disks drives and :t: would be the tape

- restore : see the iRMX manuals.

- super : prompts for password to be super user

- aedit fil_nam.ext

invokes the AEDIT fullscreen editor
- use **TAB** to view command bar at bottom of screen
- **ESC** completes an input sequence on most commands
- **^C** aborts a commmand
- use arrow keys
- **HOME** moves (page, BOL, EOL) w.r.t. the last arrow movement
- **i**(nsert)
- **d**(elete) move cursor **d**(elete) : deletes region
- **b**(uffer) move cursor **b**(uffer) : buffers region
- the current version of the file is saved under the name fil_name.bak

an amperstand '&' is used at the end of a line to continue on to the next line.
Example:

- link86 test.obj, &
**** fortran.lib to test &**
**** bind**

is equivalent to

- link86 test.obj, fortran.lib to test bind

General Comments:

The most recent crossbar file compiled in a specific directory is the one that is by default loaded into the sequencer and crossbar, not necessarily the one associated with the program being ran (i.e., when the last crossbar program was compiled the files 'seq.abs' and 'xbar.abs' were created in the current directory. If they are not renamed they are over written the next time the crossbar compiler is invoked in that directory.)

The Pascal I/O parameters PROCLIST and OUTLOG default to files with the same name, unless redirection is used at run-time.

Assume that T3 uses the descriptors \PROCLIST and OUTLOG as input and output respectively. The command

- T3

inputs from the file 'proclist' and outputs to the file 'outlog' while

- T3(proclist=T3list)

inputs from T3list and

- T3(outlog=:lp:,proclist=T3list)

writes its outlog data to the printer and inputs from the file T3list.

Input to host programs defaults to console or keyboard input, unless the program is executed from within a command file. Input to a program running within a command file has to follow the invocation of the program in the command file.

Example contents of a command file which runs the program 'test' expecting inputs '4' and '1.54':

```
test
4 1.54
```

this invokes the program 'test' which expects as input on the next line an integer and a real number.

To pass data to a command file the following format can be used.

- submit fppmu(t3.out,T3list)

where %0 is assigned to t3.out and %1 to T3list. If the invocation of FPPMU was

fppmu(outlog=%0,proclist=%1)

the fppmu host program would direct its outlog data to T3.out and its proclist input would be from T3list.

also

- submit fppmu(t3.out,T3list,12)

where in addition to the above %2 is assigned to 12 and FPPMU is invoked by

fppmu(outlog=%0,proclist=%1)
%2

would additionally allow FPPMU to input the number 12, if FPPMU was written to input one number, say the number of iterations during the run.

Note: Programs require the correct path name to the file if it is not in your current directory or a directory that is automatically searched by the system.

Other programs not commercially available:

- reset (:PFP:reset/reset or :prog:reset)

resets the system allowing the start of another run

6.2 APPENDIX B

COMPILING, LINKING, LOCATING, AND EXECUTION

Extension conventions:

*.csd	command file
*.pas	Pascal file
*.for	FORTRAN file
*.c	iC file
*.xbc	crossbar communication file
*.obj	object file - compiler output
*.fix	target processor loadable file
*.lst	compiler error listing
*.mp1	intermediate linker output file
*.mp2	intermediate locater output file
*.abs	abslolute loadable code
processor.lst	file containing processor load data

Some Pascal host file descriptor naming conventions:

proclist	processor list mapping for loading
printer	non-screen logging output device
input	console or command file input
output	screen output device

Examples:

Running a program that does not utilize the crossbar:

The Multibus Test, MU - login to the system and do the following:

- **cd :PFP:test/mu/**
- **copy processor.mu** (view loader mapping file)

.
.
.

- **copy mu.csd** (view the command file)

.
.
.

; comments
;The command file invokes RESET, SIMLOAD and MU.
; RESET does a hardware reset on the target processors and crossbar.
; SIMLOAD inputs from the PROCESSOR.LIST file the addresses of the processors and the files to load to the processors.
; MU also reads PROCESSOR.LIST and uses the addresses to start and reference the processors.
; MU is considered the host program.
;
- **submit mu** (run the PFP Multibus Test)

; slave.fix is the processor executable code
; MU.pas is the host Pascal program source
; PROCESSOR.LIST is the load/run mapping file
;

Running a program that does utilize the crossbar: The crossbar test (in turn each processor involved sends to all of the other processors via the crossbar) - login to the system and do the following:

- **cd :PFP:test/t3/**
- **copy processor.t3** (view the processor file)

.
.
.

- **copy t3.xbc** (view the crossbar file)

.
.
.

- **copy t3.csd** (view the command file)

.
.
.

; reset - resets the PFP unit
; resetxbar 4 - clears crossbar memory for Unit 4
; simload - loads the processors
; sxbload 4 - loads the sequencer and crossbar
; masseq 4 - starts the sequencer
; t3 - runs the host program
;
- **submit t3** (runs the test)

; t3.src is the host program
; t3.fix is the processor executable code
; t3.xbc is the crossbar definition file
; PROCESSOR.LST is the load/run mapping file

The following is a command file (HPascal.CSD) for use when compiling and linking a Pascal host program.
- to compile pserver.pas type '**submit :pfp:csd/hpascal(pserver)**'

Program Listing B-1. HPASCAL.CSD

```
pasc86 %0.pas large optimize(1) symbolspace(64)
```

```
link86 %0.obj, &
 :PFP:LIB/HOST.LIB, &
 :PFP:LIB/IO.LIB, &
 :LIB:ic86/clibl.lib, &
 :LIB:ic86/cfloatl.lib, &
 :LIB:pasc86/p86rn0.lib, &
 :LIB:pasc86/p86rn1.lib, &
 :LIB:pasc86/p86rn2.lib, &
 :LIB:pasc86/p86rn3.lib, &
 :LIB:ndp87/cel87.lib, &
 :LIB:ndp87/8087.lib, &
 :sd:/rmx86/lib/large.lib &
to %0 bind notype

delete %0.obj
delete %0.mp1
```

The following is a command file (Pascal.CSD) for compiling, linking, and locating a Pascal program for the target processors. - to compile integ.pas type '**submit :pfp:csd/Pascal(integ)**'

Program Listing B-2. Pascal.CSD

```
pasc86 %0.pas code large optimize(1) symbolspace(64)

link86 &
 %0.obj, &
 %1 &
 :PFP:LIB/com.lib, &
 :LIB:ic86/clibl.lib, &
 :LIB:ic86/cfloatl.lib, &
 :LIB:pasc86/p86rn0.lib, &
 :LIB:pasc86/p86rn1.lib, &
 :LIB:ndp87/cel87.lib, &
 :LIB:ndp87/8087.lib, &
 :LIB:ndp87/rtnull.lib &
 to %0.86 notype map initcode

loc86 %0.86 to %0.abs map &
 name(pfp) &
 segsize(stack(+100h)) &
 reserve( 00000H to 007FFH, 0D0000H to 0FFFFFH )&
 order(segments(??initcode)) &
 oc(purge)

:PFP:bin/fixomf86 %0.abs e0

delete %0.obj
delete %0.86
```

```
delete %0.mp1
delete %0.abs
delete %0.mp2
```

The following is a command file (FORTRAN.CSD) for compiling, linking, and locating a FORTRAN program for the target processors. - to compile integ.for type '**submit :pfp:csd/Fortran(integ)**'

Program Listing B-3 FORTRAN.CSD

```
fort86 %0.for code notype optimize(3)

link86 &
 %0.obj, &
 %1 &
 :PFP:LIB/com.lib, &
 :LIB:ic86/clibl.lib, &
 :LIB:ic86/cfloatl.lib, &
 :LIB:fort86/f86rn0.lib, &
 :LIB:fort86/f86rn1.lib, &
 :LIB:fort86/f86rn2.lib, &
 :LIB:ndp87/cel87.lib, &
 :LIB:ndp87/8087.lib, &
 :LIB:ndp87/rtnull.lib &
 to %0.86 notype map initcode

loc86 %0.86 to %0.abs map &
 name(pfp) &
 segsize(stack(+100h)) &
 reserve( 0000H to 007FFH, 0D0000H to 0FFFFFH )&
 order(segments(??initcode)) &
 oc(purge)

:PFP:bin/fixomf86 %0.abs e0

delete %0.obj
delete %0.86
delete %0.mp1
delete %0.abs
delete %0.mp2
```

The following is a command file (C.CSD) for compiling, linking, and locating a C program for the target processors. - to compile integ.c type '**submit :pfp:csd/C(integ)**'

Program Listing B-4 C.CSD

```
ic86 %0.c code large optimize(3) searchinclude( :LIB:IC86/, :PFP:INCLUDE/ )

link86 &
 %0.obj, &
 %1 &
 :LIB:ic86/estartl.obj, &
 :LIB:ic86/threadl.obj, &
 :PFP:LIB/com.lib, &
 :LIB:ic86/clibl.lib, &
 :LIB:ic86/cfloatl.lib, &
 :LIB:ndp87/cel87.lib, &
 :LIB:ndp87/8087.lib, &
 :LIB:ndp87/rtnull.lib &
 to %0.86 notype map initcode

loc86 %0.86 to %0.abs map &
 name(pfp) &
 segsize(stack(+100h)) &
 reserve( 00000H to 007FFH, 0D0000H to 0FFFFFH )&
 order(segments(??initcode)) &
 oc(purge)

:PFP:bin/fixomf86 %0.abs e0

delete %0.obj
delete %0.86
delete %0.mp1
delete %0.abs
delete %0.mp2
```

6.3 APPENDIX C

PASCAL I/O ROUTINES

The Pascal variable types supported for I/O routines are described in Table C-1

TABLE C-1 SUPPORTED I/O VARIABLE TYPES FOR PASCAL

type	alternate type name	# bits	crossbar transfer
char	CHARACTER_08BIT	8	p1:=p0.1 (p1:=p0)
record r: real; i: real; end	COMPLEX_32BIT	64	p1:=p0.4
record r: longreal; i: longreal; end	COMPLEX_64BIT	128	p1:=p0.8
boolean	LOGICAL_08BIT	8	p1:=p0.1
integer	LOGICAL_16BIT	16	p1:=p0.1
longint	LOGICAL_32BIT	32	p1:=p0.2
real	REAL_32BIT	32	p1:=p0.2
longreal	REAL_64BIT	64	p1:=p0.4
0..255	SIGNED_08BIT	8	p1:=p0.1
integer	SIGNED_16BIT	16	p1:=p0.1
longint	SIGNED_32BIT	32	p1:=p0.2
0..255	UNSIGNED_08BIT	8	p1:=p0.1
integer	UNSIGNED_16BIT	16	p1:=p0.1
longint	UNSIGNED_32BIT	32	p1:=p0.2

The Intel version of Pascal supports including files at compile time. As such we can include files that have forward reference definitions for separately compiled functions and procedures. These files have the extension '.DEF' and are listed below.

The procedures to send and receive data between target processors follow the form:

send_XXXXXXXXX(datum); and

receive_XXXXXXXXX(datum);

where XXXXXXXXX is replaced with one of the alternate type names in Table C-1 and where datum is data of type XXXXXXXXX (or the corresponding actual type). All of the sends/receives in the following com.def file listing are to be used by the programmer with the exection of the send_buffer and receive_buffer which are called by the other routines. Pascal requires the length of a string to be at least two.

The procedures to send and receive data between the target processors and the host follow the form:

output_message(message_type,message,message_size); and

input_message(message_type,message,message_size);

where message_type is one of alternate type names in Table C-1 (like XXXXXXXXX in the send/receive procedures), where message is the variable name being passed, and where message_size is the number of datums in message. The message could be a scalar or an array and if message is an array, the variable message points to the begining memory location of the array. As before the procedures input_buffer and output_buffer are called by the input_message and output_message routines.

Two functions of interest in the COM.DEF file are the **io_ready** and the **io_not_ready** functions. The host can use these boolean functions to check if a processor is sending the host a message, and a processor can use the functions to check if the host is sending the processor a message.

Program Listing C-1. Pascal COM.DEF

com.def - defines constants and such for the message passing procedures

---

```
public communication;

const {defines the types to id numbers in accordance with Table C-1}
        CHARACTER_08BIT = 0;
        COMPLEX_32BIT = 1;
        COMPLEX_64BIT = 2;
        LOGICAL_08BIT = 3;
        LOGICAL_16BIT = 4;
        LOGICAL_32BIT = 5;
        REAL_32BIT = 6;
        REAL_64BIT = 7;
        SIGNED_08BIT = 8;
        SIGNED_16BIT = 9;
        SIGNED_32BIT = 10;
        UNSIGNED_08BIT = 11;
        UNSIGNED_16BIT = 12;
        UNSIGNED_32BIT = 13;

type {defines (alternate) types as per Table C-1}
        CHARACTER_08BIT_type = char;
        COMPLEX_32BIT_type = record r: real; i: real; end;
        COMPLEX_64BIT_type = record r: longreal; i: longreal; end;
        LOGICAL_08BIT_type = boolean;
        LOGICAL_16BIT_type = integer;
        LOGICAL_32BIT_type = longint;
        REAL_32BIT_type = real;
        REAL_64BIT_type = longreal;
        SIGNED_08BIT_type = 0..255;
        SIGNED_16BIT_type = integer;
        SIGNED_32BIT_type = longint;
        UNSIGNED_08BIT_type = 0..255;
        UNSIGNED_16BIT_type = integer;
        UNSIGNED_32BIT_type = longint;
```

```
{ a variant record for generalized message passing }
        generic_type =
         record
          case word of
           CHARACTER_08BIT: ( CHARACTER_08BIT_array: array [1..256] of
                                        CHARACTER_08BIT_type );
           COMPLEX_32BIT: ( COMPLEX_32BIT_array: array [1..256] of
                                        COMPLEX_32BIT_type );
           COMPLEX_64BIT: ( COMPLEX_64BIT_array: array [1..256] of
                                        COMPLEX_64BIT_type );
           LOGICAL_08BIT: ( LOGICAL_08BIT_array: array [1..256] of
                                        LOGICAL_08BIT_type );
           LOGICAL_16BIT: ( LOGICAL_16BIT_array: array [1..256] of
                                        LOGICAL_16BIT_type );
           LOGICAL_32BIT: ( LOGICAL_32BIT_array: array [1..256] of
                                        LOGICAL_32BIT_type );
           REAL_32BIT: ( REAL_32BIT_array: array [1..256] of REAL_32BIT_type );
           REAL_64BIT: ( REAL_64BIT_array: array [1..256] of REAL_64BIT_type );
           SIGNED_08BIT: ( SIGNED_08BIT_array: array [1..256] of
                                        SIGNED_08BIT_type );
           SIGNED_16BIT: ( SIGNED_16BIT_array: array [1..256] of
                                        SIGNED_16BIT_type );
           SIGNED_32BIT: ( SIGNED_32BIT_array: array [1..256] of
                                        SIGNED_32BIT_type );
           UNSIGNED_08BIT: ( UNSIGNED_08BIT_array: array [1..256] of
                                        UNSIGNED_08BIT_type );
           UNSIGNED_16BIT: ( UNSIGNED_16BIT_array: array [1..256] of
                                        UNSIGNED_16BIT_type );
           UNSIGNED_32BIT: ( UNSIGNED_32BIT_array: array [1..256] of
                                        UNSIGNED_32BIT_type );
         end;

procedure receive_buffer( var buffer: bytes; buffer_size: word );
procedure receive_COMPLEX_32BIT( var buffer: bytes );
procedure receive_COMPLEX_64BIT( var buffer: bytes );
procedure receive_LOGICAL_08BIT( var buffer: bytes );
procedure receive_LOGICAL_16BIT( var buffer: bytes );
procedure receive_LOGICAL_32BIT( var buffer: bytes );
procedure receive_REAL_32BIT( var buffer: bytes );
procedure receive_REAL_64BIT( var buffer: bytes );
procedure receive_SIGNED_08BIT( var buffer: bytes );
procedure receive_SIGNED_16BIT( var buffer: bytes );
procedure receive_SIGNED_32BIT( var buffer: bytes );
procedure receive_UNSIGNED_08BIT( var buffer: bytes );
procedure receive_UNSIGNED_16BIT( var buffer: bytes );
procedure receive_UNSIGNED_32BIT( var buffer: bytes );

procedure send_buffer( var buffer: bytes );
procedure send_COMPLEX_32BIT( var buffer: bytes );
procedure send_COMPLEX_64BIT( var buffer: bytes );
procedure send_LOGICAL_08BIT( var buffer: bytes );
procedure send_LOGICAL_16BIT( var buffer: bytes );
```

```
procedure send_LOGICAL_32BIT( var buffer: bytes );
procedure send_REAL_32BIT( var buffer: bytes );
procedure send_REAL_64BIT( var buffer: bytes );
procedure send_SIGNED_08BIT( var buffer: bytes );
procedure send_SIGNED_16BIT( var buffer: bytes );
procedure send_SIGNED_32BIT( var buffer: bytes );
procedure send_UNSIGNED_08BIT( var buffer: bytes );
procedure send_UNSIGNED_16BIT( var buffer: bytes );
procedure send_UNSIGNED_32BIT( var buffer: bytes );


procedure io_initialize( segment: word );
function io_ready: boolean;
function io_not_ready: boolean;

procedure input_buffer( var buffer: bytes; buffer_size: word );
procedure input_message(var message_type:word; var message:bytes;
                                                var message_size: word );

procedure output_buffer( var buffer: bytes; buffer_size: word );
procedure output_message( message_type:word; var message:bytes; message_size: word );
```

The routine and variables of interest in the next .def file are the array of structures

processor

and the procedure

initialize_proc_info(var proc_list: text);.

See the example program in section 4.1.5. for an implementation example.

Program Listing C-2. Pascal SYSINFO.DEF

sysinfo.def - used to identify and start the processors involved during a execution.

```
PUBLIC sysinfo;

const
  max_processors    = 66;
  lsb               = 1;
  msb               = 16;

type
```

```
  byte         = 0..255;
  bit_pattern  = packed array [lsb..msb] of char;
  address_string = packed array [1..5] of char;
  proc_info = record
    bank       : char;
    bank_number : integer;
    segment    : word;
    page_port  : word;
    page       : byte;
    proc_type  : integer;
    seq_ctl_port : word;
    exor       : bit_pattern;
  end;

var

  number_of_proc : integer;
  address      : address_string;

  processor    : packed array [1..max_processors] of proc_info; { an array of structures
       used to store pertinent hardware mapping information}

  FUNCTION power16(exp: integer) : longint;
  FUNCTION convert_to_number : word;
  PROCEDURE get_system_parameters;  {stores hardware mapping information in the
       above array of structures 'processor'}
  PROCEDURE initialize_proc_info( var proc_list: text ); { inputs the processor list}
```

The routines of interest in the next .def file are the two procedures

turn_on_repeater(on_page_port : word; on_page : byte);

turn_off_repeater(off_page_port : word);

where on/off_page_port will in general be the page_port element of the array processor (e.g., processor[i].page_port) and the on_page will be will be the page element of the array (e.g., processor[i].page).

See the example program in section 4.1.5. for an implementation example.

Program Listing C-3. Pascal MULTIIO.DEF

multiio.def - used to access the multibus repeater system.

```
PUBLIC multi;
```

```
const
  enable_repeater  = 0100h;
  disable_repeater = 0000h;
  max_page_read    = 20;

  PROCEDURE turn_on_repeater( on_page_port : word; on_page : byte );
  PROCEDURE turn_off_repeater( off_page_port : word );
```

The routines of interest in the next .def file are the two procedures

start_all_proc;

set_up_host_io;

See the example program in section 4.1.5. for an implementation example.

Program Listing C-4. Pascal STARTPROC.DEF

startproc.def - used to initialize host to target I/O and to start involved processors.

```
PUBLIC startproc;

  PROCEDURE start_all_proc; {starts all involved processors}
  PROCEDURE set_up_host_io; {initializes status fields on all involved processors}
```

The following procedures are straight forward in their functions. (The function gotoXY uses the upper left corner of the screen as position (0,0) and the positive direction of X and Y are to the right and down respectively.

Program Listing C-5. Pascal SCREEN.DEF

screen.def - HDS terminal handling procedures.

```
PUBLIC Screen;

Procedure Clearscreen;
Procedure gotoXY(x,y : integer);
```

```
Procedure home;
Procedure beep;
Procedure clearcursor;
Procedure blockcursor;
Procedure linecursor;
```

6.4 APPENDIX D

FORTRAN I/O ROUTINES

The FORTRAN variable types supported for I/O routines are described in Table D-1

TABLE D-1 SUPPORTED I/O VARIABLE TYPES FOR FORTRAN.

type	alternate name/type	# bits	crossbar transfer
character	CHARACTER_08BIT	8	p1:=p0.1 (p1:=p0)
complex*4	COMPLEX_32BIT	64	p1:=p0.4
complex*8	COMPLEX_64BIT	128	p1:=p0.8
	LOGICAL_08BIT	8	p1:=p0.1
	LOGICAL_16BIT	16	p1:=p0.1
	LOGICAL_32BIT	32	p1:=p0.2
real*4	REAL_32BIT	32	p1:=p0.2
real*8	REAL_64BIT	64	p1:=p0.4
integer*1	SIGNED_08BIT	8	p1:=p0.1
integer*2	SIGNED_16BIT	16	p1:=p0.1
integer*4	SIGNED_32BIT	32	p1:=p0.2
	UNSIGNED_08BIT	8	p1:=p0.1
	UNSIGNED_16BIT	16	p1:=p0.1
	UNSIGNED_32BIT	32	p1:=p0.2

The Intel version of FORTRAN supports including files at compile time. These files are called have the extension '.INC' and are listed below.

The procedures to send and receive data between target processors follow the form:

send_XXXXXXXXX(datum) and

receive_XXXXXXXXX(datum)

where XXXXXXXXX is replaced with one of the alternate type names in Table D-1 and where datum is data of type XXXXXXXXX (or the corresponding actual type). All of the sends/receives in the following com.inc file listing are to be used by the programmer with the exception of the send_buffer and receive_buffer which are called by the other routines.

The procedures used to send and receive data between the target processors and the host follow the form:

output_message(%VAL(message_type),message,%VAL(message_size)) and

input_message(message_type,message,message_size)

where message_type is one of alternate type names in Table D-1 (like XXXXXXXXX in the send/receive procedures), where message is the variable name being passed, and where message_size is the number of datums in message. The message could be a scalar or an array and if message is an array, the variable message points to the begining memory location of the array. As before the procedures input_buffer and output_buffer are called by the input_message and output_message routines. See note below for the %VAL function.

Program Listing D-1. FORTRAN COM.INC

com.inc -

```
      INTEGER*2 CHARACTER_08BIT
      PARAMETER (CHARACTER_08BIT = 0)

      INTEGER*2 COMPLEX_32BIT
      PARAMETER (COMPLEX_32BIT = 1)
      INTEGER*2 COMPLEX_64BIT
      PARAMETER (COMPLEX_64BIT = 2)

      INTEGER*2 LOGICAL_08BIT
      PARAMETER (LOGICAL_08BIT = 3)
      INTEGER*2 LOGICAL_16BIT
      PARAMETER (LOGICAL_16BIT = 4)
      INTEGER*2 LOGICAL_32BIT
      PARAMETER (LOGICAL_32BIT = 5)

      INTEGER*2 REAL_32BIT
      PARAMETER (REAL_32BIT = 6)
      INTEGER*2 REAL_64BIT
      PARAMETER (REAL_64BIT = 7)

      INTEGER*2 SIGNED_08BIT
      PARAMETER (SIGNED_08BIT = 8)
      INTEGER*2 SIGNED_16BIT
      PARAMETER (SIGNED_16BIT = 9)
      INTEGER*2 SIGNED_32BIT
      PARAMETER (SIGNED_32BIT = 10)

      INTEGER*2 UNSIGNED_08BIT
      PARAMETER (UNSIGNED_08BIT = 11)
      INTEGER*2 UNSIGNED_16BIT
      PARAMETER (UNSIGNED_16BIT = 12)
      INTEGER*2 UNSIGNED_32BIT
      PARAMETER (UNSIGNED_32BIT = 13)
```

These statements are examples of sending messages of length two between the host and the target processors.

Note: FORTRAN defaults to putting the address of a variable on to the stack in a subroutine call. The %VAL() function forces FORTRAN to place the value on the stack. Also note that when passing a character string as a parameter to a subroutine in FORTRAN, the size of the string is automatically placed on to the stack after the address of the string.

Program Listing D-2. Example FORTRAN I/O Usage

```
call output_message( %VAL(CHARACTER_08BIT), 'xy' )

call output_message( %VAL(COMPLEX_32BIT), c32, %VAL(2))
call output_message( %VAL(COMPLEX_64BIT), c64, %VAL(2))

call output_message( %VAL(LOGICAL_08BIT), l08, %VAL(2))
call output_message( %VAL(LOGICAL_16BIT), l16, %VAL(2))
call output_message( %VAL(LOGICAL_32BIT), l32, %VAL(2))

call output_message( %VAL(REAL_32BIT), r32, %VAL(2) )
call output_message( %VAL(REAL_64BIT), r64, %VAL(2) )

call output_message( %VAL(SIGNED_08BIT), s08, %VAL(2) )
call output_message( %VAL(SIGNED_16BIT), s16, %VAL(2) )
call output_message( %VAL(SIGNED_32BIT), s32, %VAL(2) )

call output_message( %VAL(UNSIGNED_08BIT), u08,%VAL(2))
call output_message( %VAL(UNSIGNED_16BIT), u16,%VAL(2))
call output_message( %VAL(UNSIGNED_32BIT), u32,%VAL(2))
```

These routines are used to send messages from the host to the target processors and vice versa.

The following are examples of sending and receiveing the first element of an array of the approriate type between target processors:

Program Listing D-3. Example FORTRAN Send/Receive Usage

```
call send_COMPLEX_32BIT( c32(1) )
call send_COMPLEX_64BIT( c64(1) )

call send_LOGICAL_08BIT( l08(1) )
call send_LOGICAL_16BIT( l16(1) )
call send_LOGICAL_32BIT( l32(1) )

call send_REAL_32BIT( r32(1) )
call send_REAL_64BIT( r64(1) )

call send_SIGNED_08BIT( s08(1) )
call send_SIGNED_16BIT( s16(1) )
call send_SIGNED_32BIT( s32(1) )

call send_UNSIGNED_08BIT( u08(1) )
```

```
call send_UNSIGNED_16BIT( u16(1) )
call send_UNSIGNED_32BIT( u32(1) )

call receive_COMPLEX_32BIT( c32(1) )
call receive_COMPLEX_64BIT( c64(1) )

call receive_LOGICAL_08BIT( l08(1) )
call receive_LOGICAL_16BIT( l16(1) )
call receive_LOGICAL_32BIT( l32(1) )

call receive_REAL_32BIT( r32(1) )
call receive_REAL_64BIT( r64(1) )

call receive_SIGNED_08BIT( s08(1) )
call receive_SIGNED_16BIT( s16(1) )
call receive_SIGNED_32BIT( s32(1) )

call receive_UNSIGNED_08BIT( u08(1) )
call receive_UNSIGNED_16BIT( u16(1) )
call receive_UNSIGNED_32BIT( u32(1) )
```

6.5 APPENDIX E

C I/O ROUTINES

The C variable types supported for I/O routines are described in Table E-1

TABLE E-1 SUPPORTED I/O VARIABLE TYPES FOR C.

type	alternate name/type	# bits	crossbar transfer
char	CHARACTER_08BIT_type	8	p1:=p0.1 (or p1:=p0)
struct { float r; float i; }	COMPLEX_32BIT_type	64	p1:=p0.4
struct { double r; double i; }	COMPLEX_64BIT_type	128	p1:=p0.8
char	LOGICAL_08BIT_type	8	p1:=p0.1
short	LOGICAL_16BIT_type	16	p1:=p0.1
long	LOGICAL_32BIT_type	32	p1:=p0.2
float	REAL_32BIT_type	32	p1:=p0.2
double	REAL_64BIT_type	64	p1:=p0.4
signed char	SIGNED_08BIT_type 8	p1:=p0.1	
signed short	SIGNED_16BIT_type16	p1:=p0.1	
signed long	SIGNED_32BIT_type 32	p1:=p0.2	
unsigned char	UNSIGNED_08BIT_type	8	p1:=p0.1
unsigned short	UNSIGNED_16BIT_type	16	p1:=p0.1
unsigned long	UNSIGNED_32BIT_type	32	p1:=p0.2

The Intel version of C supports including header files at compile time. As such we can include files that have forward reference definitions for seperately compiled functions and procedures. These files are called have the extension '.h' and are listed below.

The procedures used to send and receive data between target processors follow the form:

send_XXXXXXXXX(datum); and

receive_XXXXXXXXX(datum);

where XXXXXXXXX is replaced with one of the alternate type names in Table E-1 and where datum is data of type XXXXXXXXX (or the corresponding actual type). All of the sends/receives in the following com.inc file listing are to be used by the programmer with the exception of the send_buffer and receive_buffer which are called by the other routines.

The procedures used to send and receive data between the target processors and the host follow the form:

output_message(message_type,message,message_size); and

input_message(message_type,message,message_size);

where message_type is one of alternate type names in Table E-1 (like XXXXXXXXX in the send/receive procedures), where message is the variable name being passed, and where message_size is the number of datums in message. The message could be a scalar or an array and if message is an array, the variable message points to the begining memory location of the array. As before the procedures input_buffer and output_buffer are called by the input_message and output_message routines.

Program Listing E-1. C COM.H

com.h - definitions and constants for message passing routines

```
#define CHARACTER_08BIT 0
#define COMPLEX_32BIT 1
#define COMPLEX_64BIT 2
#define LOGICAL_08BIT 3
#define LOGICAL_16BIT 4
#define LOGICAL_32BIT 5
#define REAL_32BIT 6
#define REAL_64BIT 7
#define SIGNED_08BIT 8
#define SIGNED_16BIT 9
#define SIGNED_32BIT 10
#define UNSIGNED_08BIT 11
#define UNSIGNED_16BIT 12
#define UNSIGNED_32BIT 13

typedef char CHARACTER_08BIT_type;
typedef struct { float r; float i; } COMPLEX_32BIT_type;
typedef struct { double r; double i; } COMPLEX_64BIT_type;
typedef char LOGICAL_08BIT_type;
typedef short LOGICAL_16BIT_type;
typedef long LOGICAL_32BIT_type;
typedef float REAL_32BIT_type;
typedef double REAL_64BIT_type;
typedef signed char SIGNED_08BIT_type;
typedef signed short SIGNED_16BIT_type;
typedef signed long SIGNED_32BIT_type;
typedef unsigned char UNSIGNED_08BIT_type;
typedef unsigned short UNSIGNED_16BIT_type;
typedef unsigned long UNSIGNED_32BIT_type;

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

extern void io_initialize( unsigned short );
extern unsigned char io_ready( void );
extern unsigned char io_not_ready( void );

extern void input_buffer( void *, unsigned short );
extern void input_message( unsigned short *, void *, unsigned short * );

extern void output_buffer( void *, unsigned short );
extern void output_message( unsigned short, void *, unsigned short );
```

```
extern void receive_buffer( void *, unsigned short );
extern void receive_CHARACTER_08BIT( CHARACTER_08BIT_type * );
extern void receive_COMPLEX_32BIT( COMPLEX_32BIT_type * );
extern void receive_COMPLEX_64BIT( COMPLEX_64BIT_type * );
extern void receive_LOGICAL_08BIT( LOGICAL_08BIT_type * );
extern void receive_LOGICAL_16BIT( LOGICAL_16BIT_type * );
extern void receive_LOGICAL_32BIT( LOGICAL_32BIT_type * );
extern void receive_REAL_32BIT( REAL_32BIT_type * );
extern void receive_REAL_64BIT( REAL_64BIT_type * );
extern void receive_SIGNED_08BIT( SIGNED_08BIT_type * );
extern void receive_SIGNED_16BIT( SIGNED_16BIT_type * );
extern void receive_SIGNED_32BIT( SIGNED_32BIT_type * );
extern void receive_UNSIGNED_08BIT( UNSIGNED_08BIT_type * );
extern void receive_UNSIGNED_16BIT( UNSIGNED_16BIT_type * );
extern void receive_UNSIGNED_32BIT( UNSIGNED_32BIT_type * );

extern void send_buffer( void *, unsigned short );
extern void send_CHARACTER_08BIT( CHARACTER_08BIT_type * );
extern void send_COMPLEX_32BIT( COMPLEX_32BIT_type * );
extern void send_COMPLEX_64BIT( COMPLEX_64BIT_type * );
extern void send_LOGICAL_08BIT( LOGICAL_08BIT_type * );
extern void send_LOGICAL_16BIT( LOGICAL_16BIT_type * );
extern void send_LOGICAL_32BIT( LOGICAL_32BIT_type * );
extern void send_REAL_32BIT( REAL_32BIT_type * );
extern void send_REAL_64BIT( REAL_64BIT_type * );
extern void send_SIGNED_08BIT( SIGNED_08BIT_type * );
extern void send_SIGNED_16BIT( SIGNED_16BIT_type * );
extern void send_SIGNED_32BIT( SIGNED_32BIT_type * );
extern void send_UNSIGNED_08BIT( UNSIGNED_08BIT_type * );
extern void send_UNSIGNED_16BIT( UNSIGNED_16BIT_type * );
extern void send_UNSIGNED_32BIT( UNSIGNED_32BIT_type * );
```